Compiler-assisted Memory Encryption for Embedded Processors

Nagarajan, V., Gupta, R., & Krishnaswamy, A. (2007). Compiler-assisted Memory Encryption for Embedded Processors. In Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers. (pp. 7-22). (HiPEAC'07). Berlin, Heidelberg: Springer-Verlag.

Compiler-Assisted Memory Encryption for Embedded Processors

Vijay Nagarajan, Rajiv Gupta, and Arvind Krishnaswamy

University of Arizona, Dept. of Computer Science, Tucson, AZ 85721
{vijay,gupta,arvind}@cs.arizona.edu

Abstract. A critical component in the design of secure processors is memory encryption which provides protection for the privacy of code and data stored in off-chip memory. The overhead of the decryption operation that must precede a load requiring an off-chip memory access, decryption being on the critical path, can significantly degrade performance. Recently hardware counter-based one-time pad encryption techniques [11, 13, 9] have been proposed to reduce this overhead. For high-end processors the performance impact of decryption has been successfully limited due to: presence of fairly large on-chip L1 and L2 caches that reduce off-chip accesses; and additional hardware support proposed in [13, 9] to reduce decryption latency. However, for low- to medium-end embedded processors the performance degradation is high because first they only support small (if any) on-chip L1 caches thus leading to significant off-chip accesses and second the hardware cost of decryption latency reduction solutions in [13, 9] is too high making them unattractive for embedded processors. In this paper we present a compiler-assisted strategy that uses minimal hardware support to reduce the overhead of memory encryption in low- to medium-end embedded processors. Our experiments show that the proposed technique reduces average execution time overhead of memory encryption for low-end (medium-end) embedded processor with 0 KB (32 KB) L1 cache from 60% (13.1%), with single counter, to 12.5% (2.1%) by additionally using only 8 hardware counter-registers.

1 Introduction

There is significant interest in the development of secure execution environments which guard against software piracy, violation of data privacy, code injection attacks etc. [7, 13, 11, 5, 14, 15, 12]. Memory encryption is a critical component of such secure systems. While the code and data residing on-chip are considered to be safe, code and data residing off-chip can be subject to attacks. Thus, for example, to provide defense against software piracy and to protect the privacy of data, memory encryption is employed. Data leaving the chip must be encrypted before being sent off-chip for storing and when this data is referenced again by the processor, and brought on-chip, it must be decrypted.

The overhead of decryption operations that precede loads requiring access to off-chip memory can be very high and hence can lead to significant performance degradation. Techniques based on one-time-pad encryption or counter

mode block ciphers [13, 11, 9] have been proposed that encrypt/decrypt data using a *one-time-pad* derived from a *key* and a mutating *counter* value. This enables the computation of the one-time-pad to be decoupled from the reading of encrypted data from off-chip memory. For high-end processors the performance impact of decryption has been successfully limited due to two reasons. First, the presence of fairly large on-chip L1 and L2 caches reduce the frequency of off-chip accesses. Second in [13, 9] additional hardware support has been proposed to reduce decryption latency. The techniques in [13] and [9] enable the *one-time-pad* to be precomputed while data is being fetched from off-chip memory by caching counter values and predicting them respectively. While the combination of above techniques is quite effective in limiting performance degradation in high-end processors, they rely on significant on-chip hardware resources. An additional cache is used [13] for caching the counter values. It is well known that caches occupy most of the on-chip space (about 50%) in embedded processors [18]. For this reason, several embedded processors are built without a cache as shown in Table 1. To obviate a large counter cache, the prediction technique [9] uses multiple predictions to predict the value of the counter and speculatively perform the multiple decryptions. As it performs multiple decryptions, 5 decryptions are performed in parallel, this technique requires multiple decryption units. In the prototype implementation of the AEGIS single-chip secure embedded processor [10], each encryption unit causes significant increase in the gate count (3 AES units and an integrity verification unit caused a 6 fold increase in the logic count). Thus, the hardware cost of decryption latency reduction solutions in [13, 9] is too high making them unattractive for low- to medium-end embedded processors. For such processors the performance degradation is high because first they do not support on-chip L2 data caches and they only support small (if any) on-chip L1 data caches thus leading to significant off-chip accesses. Table 1 presents a list of commercial embedded processors which have no on-chip L2 cache, many of them have no on-chip L1 data cache while others have L1 data caches varying from 2 KB to 32 KB.

Table 1. Data Cache Sizes of Embedded Processors.

D-Cache	Embedded Processor – 20 MHz to 700 MHz
0 KB	[18] ARM7EJ-S, ARM Cortex-M3, ARM966E-S [19] SecureCore SC100/110/200/210
2 KB	[20] STMicro ST20C2 50
4 KB	[16] NEC V832-143, Infineon TC1130 [16] NEC VR4181, Intel 960IT
8 KB	[20] NEC V850E-50, Infineon TC11B-96 [16] Xilinx Virtex IIPro
16 KB	[16] Motorola MPC8240, Alchemy AU 1000
32 KB	[20] MIPS 20Kc, AMD K6-2E, AMCC 440GX [20] AMCC PPC440EP, Intel XSscale Core

In this paper we develop a compiler-assisted strategy where the expensive task of finding the counter value needed for decrypting data at a load is performed under compiler guidance using minimal hardware support. Thus, the need to either cache or predict counter values using substantial on-chip hardware resources is eliminated. In addition to the global counter, our technique uses multiple compiler allocated counters for each application. The additional

counters are implemented as special registers which reside on chip. The compiler allocates these counters for store instructions. Instructions that compute the one-time-pad (using the allocated counter value) are introduced preceding the stores and loads by the compiler. For a store, the counter used is the one that is allocated to it. For a load, the counter used is the one belonging to its matching store – this is the store that frequently writes the values that are later read by the load. Through its allocation policy, the compiler tries to ensure that when a load is executed, the value of the counter allocated to its matching store has not changed since the execution of the store. Thus, the pad needed for decryption can usually be determined correctly preceding the load using the counter associated with the matching store. In other words, a prediction for the counter that is to be used for a load is being made at compile-time and hard coded into the generated code. Our experiments show that for vast majority of frequently executed loads and stores, matches can be found that produce highly accurate compile-time predictions. When a prediction fails, the value of the pad is computed using the information fetched from off-chip memory (the counter value used in encryption). The global counter is used to handle all loads and stores that cannot be effectively matched by the compiler. Our experiments show that this compiler-assisted strategy supports memory encryption at a reasonable cost: minimal on-chip resources; and acceptable code size and execution time increases.

The remainder of this paper is organized as follows. In section 2 we review how encryption/decryption is performed and motivate this work by showing the high overhead of memory encryption for low- and medium-end embedded processors. In section 3 we present our compiler-assisted strategy in detail. Experimental results are presented in section 4. Additional related work is discussed in section 5 and we conclude in section 6.

2 Background and Motivation

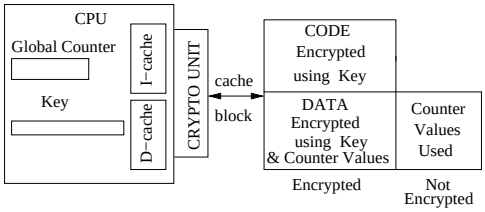

Fig. 1. Secure Processor.

Let us begin by reviewing the encryption/decryption scheme used in [11] – the organization of secure processor is shown in Fig. 1. When a plaintext data value p is to be written to off-chip memory, the corresponding ciphertext c that is actually written is computed as follows:

$$c = p \ \oplus \ Encrypt(K, Addr + Counter)$$

where '+' is the concatenation operator, K is the *key* that is kept secure inside the processor chip, *Addr* is the (block) address to which p is to be written, and

Counter is a value that is used once to produce the one-time-pad and hence is incremented after being used. A mutating counter is needed because in its absence the same pad, i.e. $Encrypt_K(Addr)$, will be repeatedly used during sequence of writes to the same memory location. Thus, a skilled attacker will be able to easily crack the ciphertexts stored in memory [13]. It should be noted that counter is needed only in case of data but not for code since code is read-only and thus never written back to off-chip memory.

Ciphertext read from the off-chip memory is decrypted by the following operation performed on-chip:

$$p = c \oplus Encrypt(K, Addr + Counter).$$

The off-chip memory is augmented with additional memory locations where the actual counter value used during encryption of data is stored. Thus, the counter value can be fetched along with the data so that decryption can be performed. However, doing so causes load latency to increase because computation of one-time-pad cannot be performed till the Counter value has been fetched. In the presence of an on-chip data cache, blocks of data are transferred across the chip boundary. All data items in a given block share the same counter value. Therefore, as described in [11], when a cache block is fetched from off-chip memory, the counter value which is one word can be fetched much faster than the larger cache block and therefore the computation of one-time-pad can be partly performed while the cache block is being fetched. This optimization takes advantage of the difference in sizes of a cache block and the counter and hence is more effective for high performance processors which have bigger cache blocks. It should be noted that the counter values are not stored in encrypted form because from these values an attacker cannot decrypt the data values stored in memory [1].

Table 2. Processor Configurations.

Low-end parameters	Medium end parameters
Processor speed : 100 MHz	Processor speed: 600 MHz
Issue: inorder	Issue: inorder
L2 cache: none	L2 cache: None
Data cache: 0, 2, 4 KB, 16B line	Data cache: 8, 16, 32 KB, 32B line
Instruction cache: 8 KB	Instruction cache: 32 KB
Memory latency (1st chunk): 12 cycles	Memory latency (1st chunk): 31 cycles
Memory bus: 100 MHz 4-B wide	Memory bus : 600 MHz 8-B wide
Load/store queue size: 4	Load/store queue size : 8
Decryption latency : 14 cycles	Decryption latency: 31 cycles
Source: [6] [20]	Source : [22]

We measured the execution time overhead of supporting memory encryption according to [11] as described above for two processor configurations: a low-end processor configuration is based upon parameters provided in [20, 6]; and a medium-end processor configuration uses parameters consistent with Intel Xscale. The decryption latency was computed based upon two real implementations of encryption and decryption hardware units described in [21] – the high performance implementation is used for medium-end processor configuration and lower performance implementation is assumed for low-end processor configuration. For low-end configuration we considered data cache sizes of 0, 2, and 4

KB while for medium-end configuration we considered data cache sizes of 8, 16, and 32 KB. The proccesor configurations are shown in Table. 2. For low-end configuration the average overhead is 60% in absence of on-chip L1 data cache while it is 12.3% and 9.9% for 2 KB amd 4 KB L1 data caches. For medium-end configuration the average overhead is 20.2%, 14.3%, and 13.1% for 8 KB, 16 KB, and 32 KB L1 data caches respectively.

3 Compiler-Assisted Encryption

In this section we show that determining the appropriate counter value can be transformed into a software (compiler) intensive task with minimal hardware support beyond the single global counter used in [11]. This transformation requires that we provide *a small number of additional on-chip counters*. These additional counters are *allocated* by the compiler to *stores* to ensure that *matching loads* (loads that read values written by the stores) can at compile-time determine which counter is expected to provide the correct counter value.

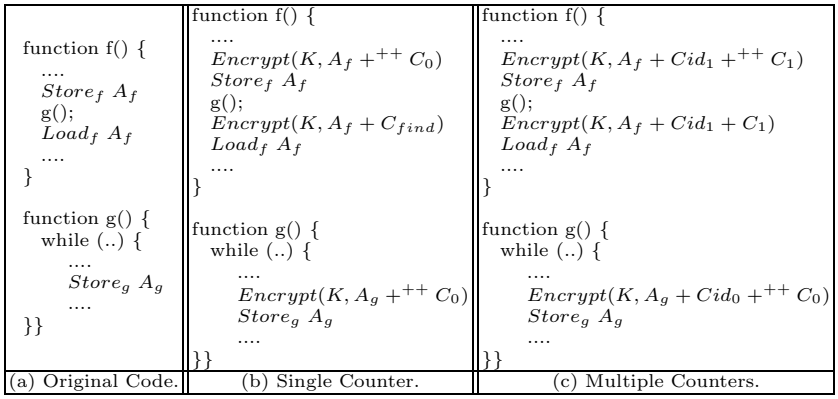

Fig. 2. Making Counter Predictable by Compile-time Allocation of Counters

We illustrate the basic principle of the above approach using the example in Fig. 2. Let us assume that function f in Fig. 2 contains a store $Store_f$ and its matching load $Load_f$ as they both reference the same address A_f. Following $Store_f$ and preceding $Load_f$ the call to function g results in execution of $Store_g$ zero or more times. For the time being let us assume that A_g can never be the same as A_f. Fig. 2b shows how the key K and counter C_0 are used preceding the stores to generate the one-time-pad by the technique in [11]. However, when we reach $Load_f$ the generation of the one-time-pad requires the counter value C_{find} – this value is fetched from off-chip memory in [11], found using caching in [13] and using prediction in [9]. Instead let us assume we are provided with additional on-chip counter-register C_1. We can use C_1 to generate one-time-pad for $Store_f$ and then regenerate this pad at $Load_f$ again by using C_1 as shown in Fig. 2c. C_1 is guaranteed to contain the correct counter value for $Load_f$ because $Store_g$ does not modify C_1 as it still relies on counter C_0. Thus by using a separate counter-register for $Store_f$ and its matching load $Load_f$, we are able to avoid

the interference created by $Store_g$. Under the assumption that A_f and A_g are never the same, the one-time-pad produced in the above manner for $Load_f$ is always correct.

From the above discussion it is clear that we need to use a separate counter-register for $Store_f$. However, as we can see, we have also introduced a prefix to the counters, *counter-id* (Cid). Next we explain the reason for adding the counter-ids. Recall that in the above discussion we assumed that A_f and A_g are never the same. Now we remove this assumption and consider the situation in which $Store_f$ A_f and $Store_g$ A_g can write to the same address. Since these stores use different counter-registers, the values of these counter-registers can be the same. When this is the case, the pads they use to write to the same address will also be the same. Obviously this situation cannot be allowed as the purpose of using counters to begin with was to ensure that pads are used only once. Hence by using a unique prefix (counter-id) for each counter, we ensure that the one-time-pads used by $Store_f$ and $Store_g$ are different even if their counter-register values match ($C_1 = C_0$) and they write to the same addresses. In other words, the counter, in our scheme is composed of the counter-id and the contents of the counter-register. Since the counter-registers are allocated at compile time, the counter-id of each allocated counter-register is known at compile time. Hence the counter-ids are hard coded into the instructions that are introduced (before stores and loads) to compute the pads.

Finally, we would like to point out that in the above example when A_f and A_g happen to be the same, it has another consequence. The pad computed at $Load_f$ in Fig. 2c will be incorrect. One way to view our technique is that it relies on the compiler to specify the identity of the counter that is *likely* to contain the needed counter value at a load but not guaranteed to contain it. When the counter value is incorrect, first we must detect that this is the case. As is the case in [11], when off-chip memory is written to, in associated storage the counter value that was used to produce the value is also stored. In our case, since each counter is prefixed by a counter-id, in addition to the counter value, the counter-id is also written. When data is fetched from memory, the counter-id and counter value are also fetched and compared with the counter-id and and counter value used to precompute the pad. If the match fails, we must recompute the pad before performing decryption; otherwise decryption can proceed right away and latency of computing the pad is hidden. Thus, incorrect precomputation of a pad does not impact the correctness of decryption but rather it affects the performance as pad computation time cannot be hidden.

The task of the compiler is to identify matching (store,load) pairs and allocate counter-registers (and counter-ids) accordingly. Finally, the counter register and counter-id assignments made to the stores and loads are communicated to the hardware through special instructions which are introduced preceding the appropriate stores and loads. For this purpose, the ISA is extended with instructions that manipulate the counter registers. We introduce two new instructions: the `incctr` instruction is used to increment a specified counter-register (C_k); and the `Counter` instruction that specifies the counter-register (C_k) and counter-id

(Cid), to use for encryption/decryption. These instructions are implemented using an unimplemented opcode in the ARM processor. Stores and loads that cannot be effectively matched are simply handled using the global counter – in other words no instructions are introduced to handle them.

Fig. 3 summarizes the modified secure processor model needed by our technique. The global counter (C_0) is used in the encryption/decryption of all data written/read by stores that do not have any matching counterparts. Counter-registers ($C_1, C_2, ...C_n$) are the counters that are allocated by the compiler to stores. Data memory contents are now encrypted and for each block of memory in it, a word of memory containing additional information is present which includes the counter-register value and the counter-id that was used to encrypt the data.

Security of the scheme: Note that the use of multiple counters does not adversely affect the security of the encryption scheme. The concatenation of the (counter-id, counter value) pair in our scheme can be considered as the unique counter and hence our compiler-assisted encryption scheme is a direct implementation of the counter-mode encryption scheme, which is proved to be a secure symmetric encryption scheme [1]. Further it is shown that the the counters may be safely leaked without compromising the security [1]. Hence the counter values and counter-ids are not stored in encrypted form.

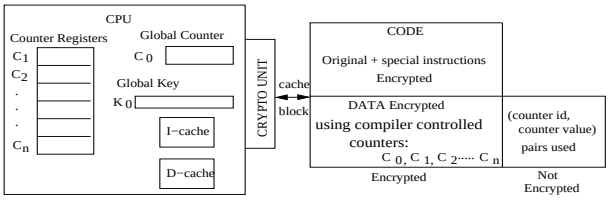

Fig. 3. Secure Processor.

We have explained the basic principles of our technique. Obviously, the number of counter-registers needed for a program to carry out the above task is an important issue, since counter-registers require additional on chip hardware. The size of the counter-id field is directly dependant on the number of counters allocated. Although counter-ids do not take up on-chip resource, it is still desirable to keep the size of this field minimal, since counter ids are to be stored along with each memory block. As our experiments show, with careful allocation of counters, it is sufficient to support 8 counter-registers and generate up to 32 counter ids (5 bits of counter-id).

In our implementation we use 32 bits to store the combined counter-id and counter value. When data value stored is encrypted using the Global Counter, we indicate the use of Global counter by having the most significant bit as 1 and remaining 31 bits represent the Global Counter Value. When data stored is encrypted using an allocated counter-register, the most significant bit is always 0, next 7 bits represent the counter-id (this restricts the maximum number of counter-ids to 128, which is more than sufficient), and the remaining 3 bytes are used to store the counter-register Value. Using 3 byte counter-register is sufficient

in our design because multiple counter- registers are used and each such register is shared by a smaller number of stores. The Global Counter, on the other hand, is shared by a larger number of stores and therefore we maintain a 31 bit Global Counter. Our experiments confirm that smaller size counters are adequate.

Next we describe the compiler algorithm used to assign counter-ids and counter-registers to selected stores and the generation of code based upon counter-registers used for matching loads. Before we present this algorithm we first discuss how opportunities for sharing counter-registers across multiple stores arise and can be exploited.

Sharing Counter-Registers. We assign different generated counter-ids to different static stores. Therefore, two stores being handled using two different counter-ids need not always use different counter values and hence different counter-registers. Two different counter-registers are necessary only if these two static stores interfere with each other as was illustrated by our example in Fig. 2. Next we show that the above observation can be exploited to enable counter-register sharing in certain cases. We describe two general categories of sharing opportunities: *intraprocedural sharing* and *across procedure sharing*. We describe these opportunities using examples given in Fig. 4.

In the first example in Fig. 4, function f contains three stores which have all been assigned different generated counter-ids (Cid_1, Cid_2, and Cid_3). Let us assume that we have a counter-register C_f which is incremented each time the function f is entered. The counter C_f's value can be used without any restriction by $Store_1$ and $Store_3$ since they are executed at most once in the function. While $Store_2$ is executed multiple times, if the compiler can determine that during each iteration of while loop address A_2 changes, then $Store_2$ can also use C_f's value that is computed upon entry to f for deriving its pad. In other words, during a single invocation of f, it is safe for all three stores to use the same value of C_f to derive their pads, as they use different counter-ids. The load at the end of the function now knows to refer to counter-register C_f irrespective of which of the three stores it is matched to and the counter-id Cid_l is one of the three depending upon which store the load is matched to based upon the profile data.

The above example showed how three different stores shared the same counter registers. But note that the three stores used unique counter-ids. The second example in Fig. 4 illustrates *across procedure sharing* of both counter-ids and counter-registers. In this example two functions are shown such that each of these functions contains a counter-register to handle its local pairs of matching stores and loads: $(Store_f, Load_f)$ in function f and $(Store_g, Load_g)$ in function g. Moreover these functions are such that in between the matching pair in one function, no function call exists that can lead to the execution of either function. In other words, execution of matching store load pair in one function cannot be *interleaved* by execution of the matching pair in the other function. Thus, when we reach a store ($Store_f$ or $Store_g$), counter C_{fg} is incremented and used and when we reach a load ($Load_f$ or $Load_g$) C_{fg}'s value can be used as it has not changed since the execution of the corresponding store. Since the execution of these functions can never be interleaved, sharing the same counter-register by

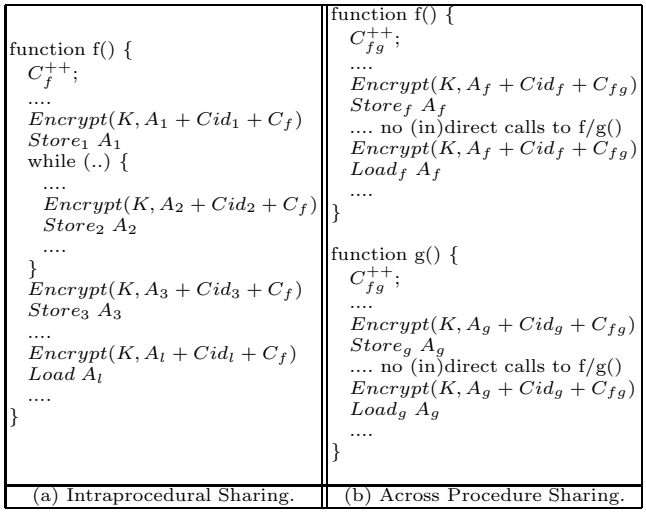

Fig. 4. Sharing Counters Across Multiple Stores.

the two functions does not lead to any interference. It should be noted that due to sharing of the counter-register, the two store load pairs in this example are guaranteed never to use the same counter value and thus they can safely share the same counter-ids. Thus *across procedural sharing* can lead to reduction of counter-ids as well as counter-registers.

In conclusion, first code can be analyzed intraprocedurally to allow sharing of counters among multiple stores in a function. Then, by analyzing the call graph and hence the lifetimes of pairs of functions, we can determine if two functions in a pair can use the same counter. Based upon the examples already presented, the conditions under which we allow sharing are as follows:

- **Intraprocedural sharing.** *Given a function, a subset of stores in the function share the same counter register, if each of the stores in the subset writes to a unique address during a single invocation of the function.* To simplify counter-register allocation, we assign at most one counter to each function and this counter is used to cover a subset of stores that satisfy the above condition. The remaining stores make use of the default global counter.
- **Across-procedure sharing.** *Given two functions, they can be assigned the same counter (counter-id, counter-register) if there does not exist a pair of store live ranges, one from each function, such that the execution of these live ranges interfere (i.e., can be interleaved with each other). Here a store live range is the program region that starts from the store instruction and extends up to and including all of its matching loads. If a function call is present within a store live range of one function that can lead to the execution of another function and hence its store live range, then the execution of two store ranges interfere with each other.* To simplify the analysis for computing interferences, we apply this optimization only to those functions whose store live ranges are local to the function (i.e., the store and all its matching loads appear within the function).

Counter-id and Counter-register Allocation Algorithm. Now we are ready to describe the steps of the complete compiler algorithm required by our technique. This algorithm operates under the constraint that we are given a certain maximum number of counter-ids (N_K) and counter-registers (N_C).

1. *Match Stores and Loads.* The first step of our algorithm is to carry out profiling and identify matching static store and load pairs. If a load matches multiple stores (as at different times during execution it receives values from different static stores) only the most beneficial matching pair is considered because in our technique a load is matched to a single store. Note that a store may match multiple loads during the above process, this is allowed by our technique.
2. *Find Counter-register Sharing Opportunities.* We look at one function at a time and identify the subset of stores in the function that satisfy the *intraprocedural counter sharing* condition given earlier. Given a function f_i, $f_i.Stores$ denotes the subset of stores identified in this step. During counter allocation in the next step, if a function f_i is allocated a counter-register, then the stores in $f_i.Stores$ will make use of that counter. For every pair of functions f_i and f_j we determine whether these functions can share the same counter according the *across procedure counter sharing* condition presented earlier. In particular, we examine the interferences among live ranges of stores in $f_i.Stores$ and $f_j.Stores$. If sharing is possible, we set $Share(f_i, f_j)$ to true; otherwise it is set to false.
3. *Allocate Counter-registers.* For each function f_i we compute a *priority* which is simply the expected benefit resulting from allocating a counter-register to function f_i. This benefit, which is computed from profile data, is the total number of times values are passed from stores in $f_i.Stores$ to their matching loads. In order of function priority, we allocate one counter per function. Before allocating a new counter-register to a function f_i, we first check if a previously allocated counter-register can be reused. A previously allocated counter-register C can be reused if $Share(f_i, f_j)$ is true for all f_j's that have been assigned counter-register C. If a previously allocated counter-register cannot be reused, a new one is allocated if one is available. After going through all the functions in the order of priority this step terminates.
4. *Assign Counter-ids.* For each counter-register C, we examine the set of functions F that have been assigned this counter-register. Each function $f \in F$ will use as many counter-ids as the number of stores in the $f.Stores$ set. However, the same counter-ids can be shared across the functions in F. Therefore the number of generated counter-ids that are needed to handle the functions in F is the maximum size of the $f.Stores$ set among all $f \in F$. The above process is repeated for all counter-registers. It should be noted that in this process we may exceed the number of counter-ids available N_K. However, this situation is extremely rare. Therefore we use a simple method to handle this situation. We prioritize the counter-ids based upon the benefits of the stores with which the counter-ids are associated. The stores corresponding to the low priority counters are then handled using the global counter such that the number of generated counter-ids does not exceed N_K.

5. *Generate Code.* All static stores that have been assigned a generated counter-id and allocated a counter register, and their matching loads, are now known. Thus we can generate the appropriate instructions for computing the pad preceding each of these instructions. All stores and loads that have not been assigned a generated counter-id and a counter register during the above process will simply make use of the global counter and thus no code is generated to handle them.

4 Experimental Evaluation

We conducted experiments with several goals in mind. First and foremost we study the effectiveness of our approach in reducing execution time overhead of memory encryption over the scenario where memory encryption is implemented using simply the global counter-register. This study is based on the low- and medium-end processor configurations with varying L1 cache sizes as described in Fig. 2. We also evaluate the effectiveness of our compiler techniques in detecting opportunities for counter sharing. Since code size is an important concern in embedded systems, we measure the static code size increase due to our technique.

Our implementation was carried out using the following infrastructure. The `Diablo` post link time optimizer [17, 4] was used to implement the compiler techniques described in the paper.

The `Simplescalar/ARM simulator` [2] was used to collect profiles and simulate the execution of modified binaries. As we mentioned earlier, the processor configurations (low and medium) from Fig. 2 are used. We use the 128 bit AES algorithm to compute the one-time-pad using a crypto unit in hardware as mentioned before.

Experiments were performed on the `MiBench` benchmark suite [6]. The small datasets from `MiBench` were used to collect profiles and the large datasets were used in the experimental evaluations. Not all the benchmarks could be used in the experiments because at this point some of them do not go through the `Diablo` infrastructure being used.

4.1 Evaluation of Sharing Optimizations

We studied the effectiveness of our counter sharing strategy by conducting the following experiment. For each program we examined the profile data and identified all of the matching store load pairs, i.e. pairs that can benefit by our technique. The number of statically distinct stores covered by these pairs represents the number of counter-ids, and hence the number of counters, that will be needed if no counter sharing is performed. Application of sharing techniques reduces this number greatly. While the intraprocedural sharing reduces the number of counter-registers, the across-procedural sharing reduces both the number of counter-ids and counter-registers. Next we present results that show the reductions in number of counter-registers and counter-ids that is achieved by our optimizations. These results are given for different thresholds, where the threshold represents the percentage of dynamic loads covered during counter allocation. Here the percentage is with respect to all dynamic loads that can derive some benefit from our technique if enough counter-registers were available.

Table 3. Number of Counter-registers Used: APS+IS:IS:Unopt.

Benchmark	Threshold of Dynamic Loads Covered		
	100%	99%	98%
bitcount	19:46:106 (20%)	3:8:40 (8%)	2:5:16 (1%)
sha	24:55:161 (15%)	2:3:25 (8%)	1:2:24 (4%)
adpcm	12:29:58 (21%)	7:19:46 (15%)	7:18:44 (16%)
fft	23:59:191 (12%)	8:20:108 (7%)	8:19:102 (8%)
stringsearch	16:34:69 (23%)	8:18:48 (17%)	8:17:45 (18%)
crc	23:52:137 (17%)	15:35:88 (17%)	12:28:54 (22%)
dijkstra	25:60:174 (14%)	6:14:76 (8%)	5:13:67 (7%)
rijndael	21:47:226 (9%)	8:17:140 (6%)	7:15:130 (5%)
jpeg	40:138:560 (7%)	10:24:217 (5%)	6:19:178 (3%)
lame	49:144:1344 (4%)	9:30:811 (1%)	7:23:660 (1%)
qsort	25:57:164 (15%)	7:15:52 (13%)	5:11:45 (11%)

In Table 3 we present the number of counter-registers that are needed for each program in following cases: (APS+IS) with both Across-Procedure and Intraprocedural Sharing, (IS) only intraprocedural sharing, and (Unopt) without sharing. In addition, the number of counter-registers used with full sharing as a percentage of counter-registers needed without any sharing is also given. As we can see, this number is computed by threshold settings of 100%, 99% and 98%. From the data in Table 3 we draw three conclusions. First the counter sharing strategy is highly effective. For example, for threshold of 100%, the number of counter-registers used after sharing ranges from 4% to 23% of the number used without sharing (on an average we have a 5 fold reduction). Second we observe that even if we set the threshold to 99%, in all cases 16 counter-registers are sufficient. Third we observe that both intraprocedural and across-procedure sharing optimizations contribute significantly although intraprocedural sharing contributes more than across-procedure sharing.

We also measured the number of counter-ids used with and without sharing. We found that across-procedure sharing results in use of 55% of number of counter-ids needed without sharing.

4.2 Execution Time Overhead

Next we conducted experiments to study the effectiveness of our strategy in *reducing* execution time overhead. We present the overheads of the two techniques: (Encrypted Optimized) which is our compiler-assisted strategy; and (Encrypted Unoptimized) which is the version in which memory encryption was performed using simply the global counter (i.e., this corresponds to [11]). Execution times were normalized with respect to the execution time of the (Unencrypted) configuration, i.e. the configuration that does not perform memory encryption. The results of this experiment are given in Fig. 5 for low-end and medium-end processor configurations respectively. For each benchmark the three bars correspond to the three cache sizes. Each bar is stacked to allow comparison of the overheads of the (Encrypted Optimized) and (Encrypted Unoptimized).

As we can see, for a low-end processor configuration, while the average overhead of (Encrypted Unoptimized) method is 60% (0 KB), 12.4% (2 KB), and 9.9% (4 KB), the overhead of our (Encrypted Optimized) method is 12.5% (0

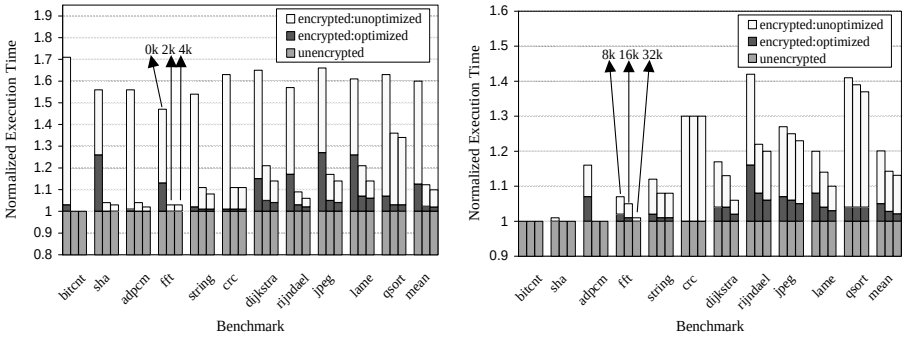

Fig. 5. Overhead of Memory Encryption - Low and Medium.

KB), 2.3% (2 KB), and 1.9% (4 KB). Thus, the benefits of using our technique are substantial for the low-end processor configuration. As expected, we observe that the benefit of our technique is greatest for processors that support no on-chip L1 cache. Moreover, our technique is beneficial for all benchmarks in this case. However, when an on-chip L1 data cache is provided, due to very low data cache miss rates, some of the benchmarks (bitcount, sha, adpcm, and fft) do not benefit from our technique. In case of the medium-end configuration the savings are substantial again (excluding the first four benchmarks that have very low miss rates). While the average overhead of (Encrypted Unoptimized) method is 20.1% (8 KB), 14.3% (16 KB), and 12.3% (32 KB), the overhead of our (Encrypted Optimized) approach is 5.0% (8 KB) 2.8% (16 KB), and 2.1% (32 KB). In the above experiment we always used 8 counter-registers and up to 32 generated counter-ids. We conducted a sensitivity study (not shown in the paper) that basically showed that the above parameters yielded the highest benefit.

Static Code Size Increase. We measured the increase in static code size due to the introduction of additional instructions. We found that the increase in the static code size was less than 1% on an average and was 3% at the maximum.

5 Related Work

We have already shown the benefits of using a small number of additional compiler controlled counter-registers over the basic technique of [11] which only uses a single global counter. As mentioned earlier, hardware enhancements to [11] have been proposed in [13] and [9]. However, significant on-chip resources are devoted for caching [13] or prediction [9] which makes these solutions unattractive for embedded processors. Memory predecryption [8] is also a technique used to hide the latency of the decryption. The basic idea here is to prefetch the L2 cache line. Prefetch can increase workload on the front side bus and the memory controller. Moreover in the absence of on-chip L1 cache prefetch would be needed for every load making this approach too expensive.

Our paper deals essentially with support for memory encryption in embedded processors which is useful for among other things for protecting the privacy of code and data in off-chip memory. Other types of attacks have also been considered by researchers which we briefly discuss next. Work has been carried out to detect tampering of information stored in off-chip memory [7, 5]. The hardware cost of detecting tampering is very high. In context of embedded processors

where on-chip resources are limited, it is more appropriate to follow the solution proposed by the commercial DS5002FP processor [23]. Tamper-detection circuitry is provided that prevents writes to be performed to off-chip memory. However, off-chip memory can be read; hence techniques are still needed to protect privacy of code and data in off-chip memory. Address leakage problem has been studied and techniques have been proposed for its prevention in [14, 15]. However, this is orthogonal to the problem we have studied. The solutions proposed in [14, 15] are still applicable. Defense against code injection attacks is also an important problem which is being extensively studied (e,g., see [3, 12]). Memory encryption techniques, such as what we have described in this paper, are also a critical component in building a defense against remote code injection attacks [3].

6 Conclusions

In this paper we argued that existing techniques for caching [13] and predicting [9] counter values for reducing memory encryption overhead, although suitable for high-performance processors, are not suitable for low- and medium-end embedded processors for which on-chip hardware resources are not plentiful. Therefore we developed a strategy in which a small number of additional counter-registers are allocated in a manner that enables that the counter-register to be used at each load is determined at compile-time. The specified counter-register is expected to contain the correct counter value needed for decryption. The only hardware cost is due to small number of counter-registers that must be supported on-chip. Our experiments show that the proposed technique reduces average execution time overhead of memory encryption for low-end (medium-end) embedded processor with 0 KB (32 KB) L1 cache from 60% (13.1%), with single counter, to 12.5% (2.1%) by additionally using only 8 compiler controlled counter-registers that accommodate 32 different counters.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work is supported by grants from Microsoft, IBM, and NSF grants CNS-0614707, CCF-0541382, and CCF-0324969 to the University of Arizona.

References

1. M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation," *38th Symposium on Foundations of Computer Science*, IEEE, 1997.
2. D. Burger and T.M. Austin, "The Simplescalar Tool Set, Version 2.0," *Computer Architecture News*, pages 13–25, June 1997.
3. C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard: Protecting Pointers from Buffer Overflow Vulnerabilities," *12th USENIX Security Symposium*, August 2003.

4. De Bus, B. De Sutter, B. Van Put, L. Chanet, D. and De Bosschere, K, "Link-Time Optimization of ARM Binaries," *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES'04).
5. B. Gassend, G. Edward Suh, D.E. Clarke, M. van Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity," *Ninth International Symposium on High-Performance Computer Architecture* (HPCA), pages 295-306, 2003.
6. M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.
7. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 168-177, November 2000.
8. B. Rogers, Y. Solihin, and M.Prvulovic, "Memory Predecryption: Hiding the Latency Overhead of Memory Encryption," *Workshop on Architectural Support for Security and Anti-Virus*, 2004.
9. W. Shi, H.-H.S.Lee, M.Ghosh, C.Lu and A.Boldyreva, "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation," *32nd Annual International Symposium on Computer Architecture* (ISCA), June 2005.
10. G.E. Suh, C.W.O'Donell, I.Sachdev, and S. Devadas "Design and Implementation of the AEGIS Single-Chip Secure Processor using Physical Random Functions" *32nd Annual International Symposium on Computer Architecture* (ISCA), June 2005.
11. G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," *36th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 339-350, December 2003.
12. N. Tuck, B. Calder, G. Varghese, "Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow," *37th Annual International Symposium on Microarchitecture* (MICRO), pages 209-220, 2004.
13. J. Yang, Y. Zhang, L. Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," *36th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 351-360, December 2003.
14. X. Zhuang, T. Zhang, and S. Pande, "HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus," *11th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 72-84, October 2004.
15. X. Zhuang, T. Zhang, H-H. S. Lee, and S. Pande, "Hardware Assisted Control Flow Obfuscation for Embedded Processors," *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (CASES), September 2004.
16. C. Zhang, F. Vahid, and W. Najjar, "A Highly Configurable Cache Architecture for Embedded Systems," *30th Annual International Symposium on Computer Architecture* (ISCA), pages 136-, 2003.
17. DIABLO, *http://www.elis.ugent.be/diablo/*.
18. http://www.arm.com/products/CPUs/embedded.html
19. http://www.arm.com/products/CPUs/securcore.html
20. Benchmark reports from http://www.eembc.org/
21. http://www.opencores.org/projects.cgi/web/ aes_core/overview/
22. Intel XScale *http://www.intel.com/design/intelxscale/*
23. DS5002FP Secure Microprocessor Chip, Dallas Semiconductor, MAXIM, *http://www.maxim-ic.com/*.